(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
11 April 2002 (11.04.2002)

PCT

(10) International Publication Number
WO 02/29602 A1

(51) International Patent Classification[7]: G06F 17/00, H04N 7/26

(21) International Application Number: PCT/AU01/01257

(22) International Filing Date: 5 October 2001 (05.10.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
PR 0634 6 October 2000 (06.10.2000) AU

(71) Applicant (*for all designated States except US*): **CANON KABUSHIKI KAISHA** [JP/JP]; 30-2, Shimomaruko 3-chome, Ohta-ku, Tokyo 146 (JP).

(72) Inventor; and
(75) Inventor/Applicant (*for US only*): **WAN, Ernest, Yiu, Cheong** [AU/AU]; 2A Wilshire Avenue, Carlingford, NSW 2118 (AU).

(74) Agent: **SPRUSON & FERGUSON**; GPO BOX 3898, Sydney, NSW 2001 (AU).

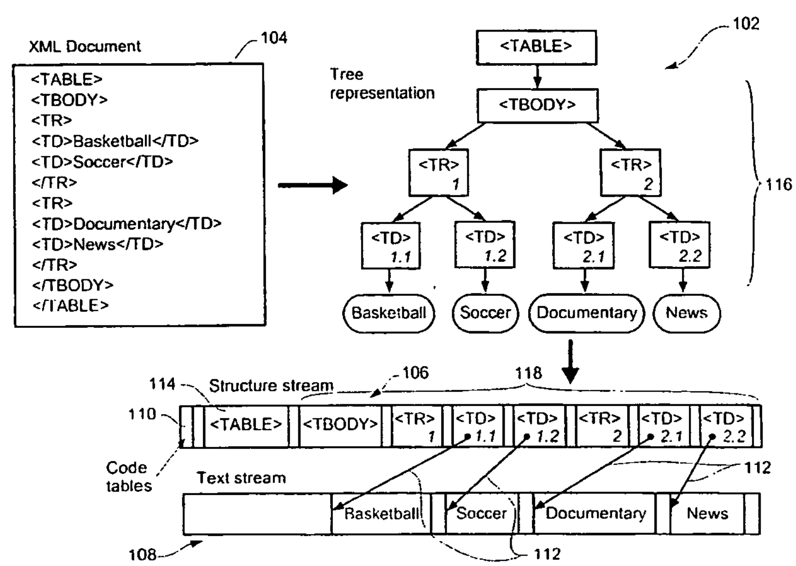
(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States (*regional*): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: XML ENCODING SCHEME



(57) Abstract: Disclosed is a method (900) for communicating at least part of a structure of a document (104) described by a hierarchical representation (102). The method identifies (902) the hierarchical representation (eg. the tree structure) of the document (104). The identification is preferably performed using XML tags. The representation is then packetized (906) into a plurality of data packets. At least one link is then created (908) between a pair of the packets, the link acting to represent an interconnection between corresponding components (eg. structure and content) of the representation. The packets are then formed (910) into a stream for communication. The links maintain the hierachical representation within the packets.

WO 02/29602 A1

XML ENCODING SCHEME

Field of the Invention

The present invention relates to the encoding of XML (Extensible Markup Language) documents and, in particular, to at least one of the compression, streaming, searching and dynamic construction of XML documents.

Background

To make streaming, downloading and storing MPEG-7 descriptions more efficient, the description can be encoded and compressed. An analysis of a number of issues relating to the delivery of MPEG-7 descriptions has involved considering the format to be used for binary encoding. Existing encoding schemes for XML, including the WBXML proposal from WAP (the Wireless Application Protocol Forum), the *Millau* algorithm and the *XMill* algorithm, have each been considered.

With WBXML, frequently used XML tags, attributes and values are assigned a fixed set of codes from a global code space. Application specific tag names, attribute names and some attribute values that are repeated throughout document instances are assigned codes from some local code spaces. WBXML preserves the structure of XML documents. The content as well as attribute values that are not defined in the Document Type Definition (DTD) can be stored in-line or in a string table. It is expected that tables of the document's code spaces are known to the particular class of applications or are transmitted with the document.

While WBXML tokenizes tags and attributes, there is no compression the textual content. Whilst such is probably sufficient for the Wireless Markup Language (WML) documents, proposed for use under the WAP, and for which WBXML is designed, as such documents usually have limited textual content, WBXML is not considered to be a very

efficient encoding format for the typical text-laden XML documents. The *Millau* approach extends the WBXML encoding format by compressing text using a traditional text compression algorithm. *Millau* also takes advantage of the schema and datatypes to enable better compression of attribute values that are of primitive datatypes.

The authors of the *Xmill* algorithm have presented an even more complex encoding scheme, although such was not based on WBXML. Apart from separating structure and text encoding and using type information in DTD and schema for encoding values of built-in datatypes, that scheme also:

(i) grouped elements of the same or related types into containers (to increase redundancy),

(ii) compressed each container separately using a different compressor,

(iii) allowed atomic compressors to be combined into more complex ones, and

(iv) allowed the use of new specialized compressors for highly specialized datatypes.

Nevertheless, existing encoding schemes are only designed for compression. They do not support the streaming of XML documents. In addition, elements still cannot be located efficiently using the XPath/XPointer addressing scheme and a document cannot be encoded incrementally as it is being constructed.

Summary of the Invention

In accordance with one aspect of the present disclosure, there is provided a method of communicating at least part of a structure of a document described by a hierarchical representation, said method comprising the steps of:

identifying said representation of said document;

packetizing said representation into a plurality of data packets, said packets having a predetermined size, said packetizing comprising creating at least one link between a pair of said packets, said link representing an interconnection between corresponding components of said representation; and

forming said data packets into a stream for communication wherein said links maintain said representation within said packets.

In accordance with another aspect of the present disclosure, there is provided a method of communicating at least part of the structure of a document described by a hierarchical representation, said method comprising the steps of:

identifying at least one part of said representation and packetizing said parts into at least one packet of predetermined size, characterised in that where any one or more of said parts of said representation do not fit within one said packet, defining at least one link from said one packet to at least one further said packet into which said non-fitting parts are packetized, said link maintaining the hierarchical structure of said document in said packets.

In accordance with another aspect of the present disclosure, there is provided a method of facilitating access to the structure of an XML document, said method comprising the steps of:

identifying a hierarchical representation of said document;

packetizing said representation into a plurality of packets of predetermined packet size;

forming links between said packets to define those parts of said representation not able to be expressed within a packet thereby enabling reconstruction of said representations after de-packetizing.

The presently disclosed encoding and decoding schemes separate structure and text encoding and use the schema and datatypes for encoding values of built-in datatypes. In addition, the disclosure provides support for streaming and allows efficient searching using XPath/XPointer-like addressing mechanism. Such also allows an XML document to be encoded and streamed as it is being constructed. These features are important for broadcasting and mobile applications. The presently disclosed encoding scheme also supports multiple namespaces and provides EBNF definitions of the bitstream and a set of interfaces for building an extensible encoder.

Brief Description of the Drawings

One or more embodiments of the present invention will now be described with reference to the drawings and Appendix, in which:

Fig. 1 schematically depicts an encoded XML document;

Fig. 2 depicts the organization of the structure segment;

Fig. 3 schematically depicts the encoder model;

Fig. 4 schematically depicts the decoder model.;

Fig. 5 schematically illustrates the encoder encoding an XML document incrementally into multiple packets;

Figs. 6A and 6B show how node locators are used for linking a node to its sub-trees in other structure packets and how each node locator contains the packet number of a sub-tree's packet;

Fig. 7 schematically depicts how a long string is stored as string fragments in multiple text packets which each packet pointing to the text packet that contains the next fragments;

Fig. 8 is a schematic block diagram representation of a computer system with which the described arrangements may be implemented;

Fig. 9 is flowchart of a XML document encoding operation;

Fig. 10 is flowchart illustrating how different data types can be handled in the encoding operations; and the

Appendix provides a definition useful for the encoded bitstream and the parameters thereof.

Detailed Description

The methods of encoding and decoding XML documents to be described with reference to Figs. 1 to 7 and 9 and 10 are preferably practiced using a general-purpose computer system 800, such as that shown in Fig. 8 wherein the processes of Figs. 1 to 7 may be implemented as software, such as an application program executing within the computer system 800. In particular, the steps of the methods may be effected by instructions in the software that are carried out by the computer. The software may be divided into two separate parts; one part for carrying out the encoding/decoding methods; and another part to manage the user interface between the encoding/decoding methods and the user. The software may be stored in a computer readable medium, including the storage devices described below, for example. The software is loaded into the computer from the computer readable medium, and then executed by the computer. A computer readable medium having such software or computer program recorded on it is a computer program product. The use of the computer program product in the computer preferably effects an advantageous apparatus for encoding/decoding XML documents.

The computer system 800 comprises a computer module 801, input devices such as a keyboard 802 and mouse 803, output devices including a printer 815 and a display

device 814. A Modulator-Demodulator (Modem) transceiver device 816 is used by the computer module 801 for communicating to and from a communications network 820, for example connectable via a telephone line 821 or other functional medium. The modem 816 can be used to obtain access to the Internet, and other network systems, such as a Local Area Network (LAN) or a Wide Area Network (WAN). A seen, a server computer system 850 connects to the network 820 enabling communications with the computer system 800. The server computer 850 typically has a similar structure and/or is operable in a like or complementary fashion to the computer system 800. For example, whilst the computer system 800 may perform an XML encoding function, the server computer 850 may perform a complementary XML decoding function, and vice versa.

The computer module 801 typically includes at least one processor unit 805, a memory unit 806, for example formed from semiconductor random access memory (RAM) and read only memory (ROM), input/output (I/O) interfaces including a video interface 807, and an I/O interface 813 for the keyboard 802 and mouse 803 and optionally a joystick (not illustrated), and an interface 808 for the modem 816. A storage device 809 is provided and typically includes a hard disk drive 810 and a floppy disk drive 811. A magnetic tape drive (not illustrated) may also be used. A CD-ROM drive 812 is typically provided as a non-volatile source of data. The components 805 to 813 of the computer module 801, typically communicate via an interconnected bus 804 and in a manner which results in a conventional mode of operation of the computer system 800 known to those in the relevant art. Examples of computers on which the described arrangements can be practised include IBM-PC's and compatibles, Sun Sparcstations or alike computer systems evolved therefrom.

Typically, the application program is resident on the hard disk drive 810 and read and controlled in its execution by the processor 805. Intermediate storage of the program and any data fetched from the network 820 may be accomplished using the semiconductor memory 806, possibly in concert with the hard disk drive 810. In some instances, the application program may be supplied to the user encoded on a CD-ROM or floppy disk and read via the corresponding drive 812 or 811, or alternatively may be read by the user from the network 820 via the modem device 816. Still further, the software can also be loaded into the computer system 800 from other computer readable medium including magnetic tape, a ROM or integrated circuit, a magneto-optical disk, a radio or infra-red transmission channel between the computer module 801 and another device, a computer readable card such as a PCMCIA card, and the Internet and Intranets including e-mail transmissions and information recorded on Websites and the like. The foregoing is merely exemplary of relevant computer readable mediums. Other computer readable media may alternately be used.

In operation the XML document encoding/decoding functions are performed on one of the server computer 850 or the computer system 800, and the packetized bit stream so formed transmitted over the communications network 820 for reception and decoding by the computer system 800 or server computer 850 respectively, as the case may be. In this fashion an XML document may be conveniently communicated between two locations in an efficient manner whilst affording optimal time at the receiver to decode the document on-the-fly as it is received without a need to first receive the entire document.

The methods of encoding and decoding may alternatively be implemented in part or in whole by dedicated hardware such as one or more integrated circuits performing the functions or sub functions of encoding and/or decoding. Such dedicated hardware may

include graphic processors, digital signal processors, or one or more microprocessors and associated memories.

Encoding and Compressing XML

Separating structure and text

Traditionally, XML documents are mostly stored and transmitted in their raw textual format. In some applications, XML documents are compressed using some traditional text compression algorithms for storage or transmission, and decompressed back into XML before they are parsed and processed.

According to the present disclosure, another way for encoding an XML document is to encode the tree hierarchy of the document (such as the DOM representation of the document). The encoding may be performed in a breadth-first or depth-first manner. To make the compression and decoding more efficient, the XML structure, denoted by tags within the XML document, can be separated from the text of the XML document and encoded. When transmitting the encoded document, the structure and the text can be sent in separate streams or concatenated into a single stream.

As seen in Fig. 1 and according to the instant embodiment, a tree representation 102 of an XML document 104, which is typically available from memory, includes a number of nodes 116 and is encoded in a depth-first fashion. The structure of the document 104 and the text contained therein can be encoded as two separate streams 106 and 108 respectively as shown, or concatenated into a single stream. The structure stream 106 is headed by the code tables 110 and 114. The encoded nodes 118 of the tree 102 each have a size field (not illustrated) that indicates the size of the node and includes the total size of its descendant nodes. Some of the encoded leaf nodes 118 contain links 112 that link those leaf nodes to their corresponding encoded content in the

text stream 108. Each encoded string in the text stream 108 is headed by a size field (not illustrated) that indicates the size of the string. Where concatenated into a single stream, packets containing the root of the links 112 should precede those packets containing the text pointed to by the links 112, thereby ensuring that the structure component of the document 104 is received by the decoder before the corresponding text (content) component.

The approach shown in Fig. 1 is also depicted in Fig. 9 as a flowchart of an encoding method 900 which may be implemented as a software program running on the computer system 800. The method 900 communicates at least part of a structure of a document described by a hierarchical representation and has an entry step 902. Initially, at step 904, the method 900 identifies the hierarchical representation (eg. the tree structure) of the document 104. The identification is preferably performed using the XML tags as mentioned above. With this, at step 906 the representation is packetized into a plurality of data packets. At step 908, at least one link is created between a pair of the packets. The link acting to represent an interconnection between corresponding components (eg. structure and content) of the representation. In step 910, the packets are formed into a stream for communication. The links maintain the hierarchical representation within the packets. The method 900 ends at step 912.

In general, the volume of structural information is much smaller than that of textual content. Structures are usually nested and repeated within a document instance. Separating structure from text allows any repeating patterns to be more readily identified by the compression algorithm which, typically, examines the input stream through a fixed-size window. In addition, the structure and the text streams have rather different

characteristics. Hence, different and more efficient encoding methods may be applied to each of the structure and text.

The structure is critical in providing the context for interpreting the text. Separating structure and text in an encoder allows the corresponding decoder to parse the structure of the document more quickly thereby processing only the relevant elements while ignoring elements (and descendants) that it does not know or require. The decoder may even choose not to buffer the text associated with any irrelevant elements. Whether the decoder converts the encoded document back into XML or not depends on the particular application to be performed (see the discussion below on Application Program Interfaces –API's).

Code tables

The elements of a document description and their attributes are defined in DTD's or schemas. Typically, a set of elements and their associated attributes are repeatedly used in a document instance. Element names as well as attribute names and values can be assigned codes to reduce the number of bytes required to encode them.

Typically, each application domain uses a different set of elements and types defined in a number of schemas and/or DTD's. In addition, each schema or DTD may contain definitions for a different namespace. Even if some of the elements and types are common to multiple classes of applications, they are usually used in a different pattern. That is, an element X, common to both domains A and B, may be used frequently in domain A, but rarely in domain B. In addition, existing schemas are updated and new schemas are created all the time. Hence, it is best to leave the code assignment to organisations that overlook interoperability in their domains. For instance, MPEG-7 descriptions are XML documents. MPEG may define the codespaces for its own

descriptors and description schemes as well as external elements and types that are used by them. MPEG may also define a method for generating codespaces. Ideally, the method should be entropy based – that is, based on the number of occurrences of the descriptors and description schemes in a description or a class of description (see the section on generating codespaces).

Separating element and attributes

An XML tag typically comprises an element name and a set of attribute name/value pairs. Potentially, a large set of attributes can be specified with an element instance. Hence, separating an element name from the attributes will allow the document tree to be parsed and elements to be located more quickly. In addition, some attributes or attribute name/value pairs tend to be used much more frequently than the others. Grouping attribute name, value and name/value pairs into different sections usually results in better compression.

Encoding values of built-in datatypes and special types

The encoder operates to encode the values of attributes and elements of built-in (or default) datatypes into more efficient representations according to their types. If the schema that contains the type information is not available, the values are treated as strings. In addition, if a value (for instance, a single-digit integer) is more efficiently represented as a string, the encoder may also choose to treat it as string and not to encode it. By default, strings are encoded as a Universal Text Format (UTF-8) string which provides a standard and efficient way of encoding a string of multi-byte characters. In addition, the UTF string includes length information avoiding the problem of finding a suitable delimiter and allowing one to skip to the end of the string easily.

Special *type encoders* can be used for special data types. These special type encoders can be specified using the setTypeEncoder() interface of the Encoder API (as discussed below). Information about the special type encoders is preferably stored in the header of the structure segment, advantageously as a table of type encoder identifiers. Further, the default type encoders (for the built-in datatypes) can be overridden using the same mechanism. As such where some built-in data type would ordinarily be encoded using a default encoder, a special encoder may alternatively be used, such necessitating identification within the bitstream that an alternative decoding process will be required for correct reproduction of the XML document. Each encoded value is preceded by the identifier of the type encoder that was used to encode the value.

In this fashion, an XML document encoder implemented according to the present disclosure may include a number of encoding formats for different types of structure and text within the XML document. Certain encoding formats may be built-in or default and used for well known or commonly encountered data types. Special type encoders may be used for any special data types. In such cases, an identification of the particular type encoder(s) used in the encoding process may be incorporated into the header of a packet, thereby enabling the decoder to identify those decoding processes required to be used for the encoded types in the encoded document. Where appropriate, the particular type encoders may be accessible from a computer network via a Uniform Resource Indicator (URI). Where the decoder is unable to access or implement a decoding process corresponding to an encoded type encountered within a packet in the encoded document, a default response may be to ignore that encoded data, possibly resulting in the reproduction of null data (eg. a blank display). An alternative is where the decoder can operate to fetch the special type decoder, from a connected network, for example using a URI that may

accompany the encoded data. The URI of an encoder/decoder format may be incorporated into the table mentioned above and thereby included in the bitstream (see the Appendix).

In a further extension of this approach, multiple encoding formats may be used for to a single data type. For example, text strings may be encoded differently based upon the length of the string, such representing a compromise between the time taken to perform a decoding process and the level of compression that may be obtained. For example, text strings with 0-9 characters may not be encoded, whereas strings with 10-99 and 100-999 characters may be encoded with respective (different) encoding formats. Further, one or more of those encoding formats may be for a special data type. As such the encoder when encoding text strings in this example may in practice use no encoding for 0-9 character strings, a default encoder for 10-99 character strings, and a special encoder for string having more than 100 text characters.

Fig. 10 shows an example of a method 1000 of encoding an XML document, that has an entry point of step 1102. Initially, at step 1004, the method 1000 examines the XML document 104to identify each data type forming part of the XML document 104. At step 1006, the method 1000 operates to identify a first set of the data types for which a corresponding special encoding format is available. Having identified the special data types, step 1008 encodes each part of the XML document having a data type in the first set with the corresponding special encoding format. Next, in step 1010, the method 1000 encodes each remaining part of the XML document with a default encoding format corresponding to the data type of the remaining part. In step 1012, a representation is formed of the information referencing at least each of the data types in the first set with the corresponding special encoding format. In step 1014, the representation is associated with

the encoded parts as an encoded form of the XML document 104. The method 1000 then ends at step 1016.

The Structure Segment (or Structure Stream)

Fig. 2 shows the various sections of the structure segment (or stream) 106. The structure segment begins with a header 202 and its body is divided into a number of sections 204. The header 202 identifies the version of the XML and that of the encoding format.

Each section 204 in the body begins with a unique signature indicating the section type. Hence, it is not necessary for the various sections to be arranged in a particular order. Nevertheless, in the following discussion, we assume the sections to be arranged in the order shown in Fig. 2. The section signature is followed by a *size* field which indicates the size of the section.

An ID table section 206 allows elements with ID's to be located quickly in a document hierarchy section 208. The ID table 206 may be absent from an encoded document even if the document has elements with ID's. This is because the DTD's or schema which contain the ID definition may not be available at the time of encoding.

A section 210 is preferably reserved for the document type declaration and the internal (DTD) subset. For XML Schema-based documents, for example MPEG-7 descriptions, this section 210 will be absent.

There are sections for the code tables for namespaces 212, element names 214, attribute names 216 and attribute values 218. Hereafter these code tables will be referred to as *local code tables* to differentiate them from any code tables that are pre-defined for both the encoder and decoder and are not carried in the bitstream. For instance, there may be *pre-defined code tables* for MPEG-7 or XML Schema.

The local code tables are usually followed by a section containing a table of attribute name/value pairs 220 which makes use of the codes defined in the local code tables as well as any pre-defined code tables.

The document hierarchy section 208 is the encoded tree structure of the XML document using codes from the local and the pre-defined code tables.

Apart from using code tables and type encoders for encoding, in most cases, the encoder also compresses each section using a *compressor*. Instead of compressing each section of the body of the structure segment 106 independently, the body of the structure segment can be compressed together. This may actually result in better compression ratio due to lesser overhead and the larger amount of data. However, such compression requires one to decompress the whole structure body in order to find out whether a document contains a particular element. Both approaches may be tested to determine which works better in practice. Nevertheless, if a section is small, compression may not be effective and the encoder may choose not to compress the section. Each section has a compressed flag to signal whether compression has been applied. If compression has been applied, the size field at the beginning of each section indicates the compressed (rather than the uncompressed) size of the section in bytes.

Potentially, a different compressor can be used for each section taking into account the characteristics of the data in each section. Information about the compressors used is provided in the header. The default is to use ZLIB for compressing all the sections in the structure segment as well as the text segment. The ZLIB algorithm generates a header and a checksum that allow the integrity of the compressed data to be verified at the decoder end.

<u>The Text Segment (or Text Stream)</u>

The text segment 108 begins with a text segment signature followed by a size field that indicates the size of the encoded text. The text segment contains a sequence of UTF-8 strings which are the text of the elements.

The Encoder and Decoder Models

The Encoder Model

Fig. 3 shows an XML encoder model 300 incorporating an encoder 302 for encoding the XML document 104 into a bitstream 306 for storage or transmission. The encoder model 300 may be implemented as a software program or sub-programs operating within the computer module 801, the program being typically stored in the HDD 810 and read and controlled in its execution by the processor 805. The bitstream 306 may be transmitted upon creation via the I/O interface 808 and network 820 for complementary decoding and reproduction by the server computer 850. Alternatively, the bitstream 306 may be stored in the HDD 810 or as a CD-ROM in the drive 812 for subsequent reproduction. The encoder 302 may support an Application Program Interface (API) 308 (eg. the DOM API) so that the document tree 102 can be encoded as the tree 102 is being created. A standard library 310 (for XML) is used to provide code tables 312, encoders 314 for built-in datatypes, and default compressors 316 that may be used in the encoding processes. Domain-specific libraries 318 may also be defined for various domains. Each domain-specific library 318 may contain code tables 320 for the particular domain and encoders 322 for some data types. An application can also provide specific modules 324 including application-specific encoders 326 for special data types as discussed above and corresponding compressors 328. However, these type encoders 326 and compressors 328 have to be either downloadable and platform-independent or pre-installed at the decoder end. An application can also instruct the encoder 326 to use its

pre-defined code tables 330. The code tables 330 can be incorporated into the bitstream 306 or pre-installed at the decoder end. Each of the individual encoders and compressors shown in Fig. 3 may be implemented by software (sub)programs or, in some instances special purpose hardware (eg. for fast encoding).

The Decoder Model

Fig. 4 shows a complementary XML decoder model 400 including a decoder 402 for decoding the XML bitstream 306 to output an XML document 104. Alternatively, the decoder may support an API 408 (eg. the SAX ("simple API for XML") or DOM API) that allows an application to construct its own internal model of the document tree 102. This saves the decoder 402 from outputting the XML document 104 and the application from re-parsing the reconstructed XML document 104. In either case, the decoder 402 uses the standard library 410, any domain-specific libraries 418 as well as any pre-installed or downloaded application-specific modules 424 (that were used by the encoder) when decoding the XML bitstream 306. In Fig. 4, elements of the decoder model 400 are numbered in a similar fashion to that of Fig. 3, such that where a difference of 100 exists in the numbering, the elements have corresponding like functions. The decoder model 400 may for example be implemented within the computer module 801 to decode the bitstream 306 received via the network 820 from the server computer 850. Alternatively, the decoder model 400 may operate to decode a bitstream obtained from the CD-ROM, for example. Like the encoder 302, software and hardware decoding processes may be used within the decoder 402.

In most cases, the decoder 402 at the client end need not validate the decoded XML document 104 of Fig. 4 against their DTD's or schemas. Validation at the client side is costly, inefficient and most likely redundant. The decoder 104 may assume that the

XML documents have been validated against their DTD's or schemas at the server end. Similarly, the underlying transport as well as any error detection mechanism such as checksums that is built into the binary format should be capable of catching any transmission error.

Locating Elements

XML elements can be referenced and located using ID's or XPath/XPointer fragments. As mentioned earlier, the ID table 206 of the structure segment 106 allows elements with ID's to be located quickly in the document hierarchy section 208. Any text and attributes associated with an element can then be located efficiently using the locators in the encoded elements.

Below are some examples of XPath fragments that can be appended to an Uniform Resource Indicator (URI):

- /doc/chapter[2]/section[3]

  selects the third section of the second chapter of doc

- chapter[contains(string(title),"Overview")]

  selects the chapter children of the context node that have one or more title children containing the text "Overview"

- child::*[self::appendix or self::index]

  selects the appendix and index children of the context node

- child::*[self::chapter or self::appendix] [position()=last()]

  selects the last chapter or appendix child of the context node

- para[@type="warning"]

  selects all para children of the context node that have a type attribute with value "warning"

• para[@id]

selects all the para children of the context node that have an id attribute.

An XPath/XPointer fragment consists of a list of location steps representing the absolute or relative location of the required element(s) within an XML document. Typically, the fragment contains a list of element names. Predicates and functions may be used, as in the examples above, to specify additional selection criteria such as the index of an element within an array, the presence of an attribute, matching attribute value and matching textual content.

The compactness of the encoded document hierarchy allows it to be parsed (and instantiated) without expanding into a full object tree representation. The fragment address is first translated into an encoded form. One of the consequences of such a translation process is that it allows one to determine immediately whether the required element(s) actually occurred in the document. Matching the components of the encoded fragment address is also much more efficient than matching sub-strings. The design allows simple XPath/XPointer fragments (which are most frequently used) to be evaluated quickly. Searching the document hierarchy first also greatly narrows the scope of subsequent evaluation steps in the case of a more complex fragment address.

Packetizing the Bitstream for Streaming

Streaming XML

Traditionally, XML documents are mostly stored and transmitted in their raw textual format. In some applications, XML documents are compressed using some traditional text compression algorithms for storage or transmission, and decompressed back into XML before they are parsed and processed. Although compression may greatly reduce the size of an XML document, under such circumstances an application still must receive the entire XML document before parsing and processing can be performed.

Streaming an XML document implies that parsing and processing can start as soon as sufficient portion of the XML document is received. Such capability will be most useful in the case of a low bandwidth communication link and/or a device with very limited resources.

Because an ordinary XML parser expects an XML document to be well-formed (ie. having matching and non-overlapping start-tag and end-tag pairs), the parser can only parse the XML document tree in a depth-first manner and cannot skip parts of the document unless the content of the XML document is reorganized to support it.

Packetizing the Bitstream

Encoding an XML document into a complete structure segment 106 and a complete text segment 108 as described earlier will greatly reduce the size of the data and, at the same time, allow some transmission error to be detected. Nevertheless, the decoder 402 still has to receive a large amount of the encoded data before it can process it. For instance, the decoder 402 will have received the code tables 110 in their entirety before parsing of the document hierarchy can commence. At the same time, the decoder 402 has to wait for the arrival of certain segment of the text segment 108 to get the text that is associated with a node. To allow processing to be started as soon as possible at the decoder end, the XML document 104, as seen in Fig. 5, has to be encoded incrementally allowing small packets 502 of encoded data 500 to be sent to the decoder 402 as they become available. In Fig. 5, the cross-hatched packets 504 denote structure packets and the diagonal-lined packets 506 denote text packets. These packets are preceded by a header packet 508 and followed by a trailer packet 510. In the preferred arrangement, each data packet 502 has the same structure as a complete structure segment 106 or a complete text segment 108. At the same time, each packet 502 may be dependent on those

packets 502 sent before it or, in some implementations, on a predetermined number of packets sent after it. Such a predetermine number may be determined dynamically.

Apart from the need for processing a document while it was being delivered, an encoder/decoder typically has an output/input buffer of fixed size. Accordingly, except for very short documents, the encoder 302 has to encode an XML document incrementally into multiple packets. Each of the packets 502 (including 504, 506, 508 and 510) is headed by a packet header. The packet header contains a *packet number* that is used as a packet ID as well as for ordering the packets and detecting any missing packets. The packet header also contains a *size* field which indicates the size of the packet 502 in bytes and a *type* field which indicate whether the packet is a structure packet 504, a text packet 506, a header packet 508, a trailer packet 510 or a further type of packet 502, named a command packet, not illustrated in Fig. 5, but described later in this document.

For each structure packet 504, the ID table incorporated therein contains only the ID's of those elements included in the packet. Its code tables contains only new codes that have not been transmitted. Codes that have been transmitted will not be re-assigned or re-mapped. The default implementation simply appends new value to the table and uses the index (augmented by the base index of the table) of the entries as their codes. A slightly more complicated (but more code efficient) method is to count the number of occurrences of the values and remapped the codes so that values that occur more frequently are remapped to shorter codes just before the packets are output. If a pre-defined code table is used or if the remapping is not based on the number of occurrences, sorting the values before compressing may result in better compression rate. A different algorithm for assigning code can be implemented. Nevertheless, once output, the codes are fixed and cannot be re-assigned to other values or re-mapped in subsequent packets. Pre-defined

code tables can also be specified using the UseCodeTable() method of the Encoder Interface described later in this specification. The method also allows one to specify whether the pre-defined code table is to be encoded with the data into the bitstream. The code tables of a number of namespaces which are fundamental to XML (or an application domain such as MPEG-7) are expected to be hardwired to all XML (MPEG-7) encoders and decoders and need not be encoded into the bitstream.

If an ID, an element name, an attribute name, or an attribute value is longer than a pre-defined length, it will be encoded in a text packet and a *string locator* rather than the actual string will appear in the tables.

The document hierarchy section of a structure packet contains a sequence of nodes. Each node has a *size* field that indicates its (encoded) size in bytes including the total size of its descendant nodes encoded in the packet. The node can be an *element node*, a *comment node*, *a text node* or a *node locator*. Each node has a *nodeType* field that indicates its type.

The document hierarchy may contain:

(i) a complete document tree: this is only possible for very short document;

(ii) a complete sub-tree: the sub-tree is the child of another node encoded in an earlier packet; and

(iii) an incomplete sub-tree: the sub-tree is incomplete because the whole sub-tree cannot be encoded into one packet due to time and/or size constraints.

Node locators are used in the manner shown in Fig. 6A, for a tree structure 622 which has incomplete sub-trees 602 and 604, for locating the missing nodes and the descendants of the incomplete sub-trees. In this regard, and with reference to the earlier example, whilst the hierarchical tree-representation 102 of the document 104 is known

when encoding takes place, upon decoding of the communicated packets, only portions of the tree representation 102 will typically be made available. As more packets are received the tree may be reconstructed. For example, in the data stream shown in Fig. 6B, a packet 620 (being the #2 packet in the data stream in this example) includes part of the tree structure 622 of a document, that structure including nodes A, B1, B2 and B3. However, in this example, the size of the packet 620 is insufficient to describe the entire tree structure 622 and to accommodate other nodes, such as B4 and D1. Node locators 608 and 606 respectively are thus incorporated into the descriptions of the corresponding parent nodes (B3 and B2 respectively) and contain the respective packet numbers 610 and 612 of a structure packets that contains a sequence of missing nodes and their sub-trees. As such, on receiving the sequence of packets illustrated in Fig. 6B, part of the tree 622 can be reconstructed upon receiving the packet (#2) 620 and the branch including node D1 can be reconstructed upon receiving packet (#7) 610 and the balance of the tree reconstructed upon receiving packet (#20) 612.

Each element node preferably contains a namespace code, an element (name) code, and, if the element has attributes, the byte offset of the first attribute in the attribute name/value pair table and the number of attributes.

Each text node or comment node typically contains a text locator rather than the actual text. The text locator specifies the packet number of a text packet and a byte offset into the text packet.

In some cases, a string may exceed the maximum size of a packet. Where such occurs, the string is stored as fragments over multiple text packets, as shown in Fig. 7. Each text packet 702 has a flag 704 indicating whether it contains a list of UTF-8 encoded strings and string locators or a string fragment. In the case of a string fragment, the packet

number of the next fragment is also included. If a text packet contains the last (or the only) fragments of a string, the packet number for the next fragment is set to zero, as shown.

Commands for Constructing Document Tree

An XML document may be packetized for streaming to the receiver as it is being encoded or even generated (according to some pre-defined DTD or schema). In this case, the XML document is typically constructed in real-time using an API such as a DOM API. Instead of parsing an XML file, the encoder 302 operates to construct the bit stream 306 from the memory representation directly. Nodes and sub-trees inserted and appended using the API are encoded as (binary) command packets to modify the memory representation at the decoder end. The packet number ensures that the command packets are executed in the correct sequence.

Since the nodes transmitted are parts of the same document (that conforms to some pre-defined DTD or schema) and the document is on-line and in-sync between the encoder 302 and decoder 402 all the time, there should not be any consistency issue in relation to the content of the nodes. In some presentations, certain information has only temporal relevance. That is, some information is only relevant within a certain period of time during the presentation. Information units (for example, the score of a football match) that are relevant to two different time instances of the presentation may themselves be inconsistent. A presentation description scheme is desirable to establish the timing and synchronization model of a presentation. The timing of any media object including XML data can be indicated by a start time and a duration. Such a presentation encoder/decoder pair would typically include an XML encoder/decoder as described above arranged internally. The presentation decoder, rather than the XML decoder, operates to interpret

the start time and duration attributes. The presentation encoder also decides whether or not to remove from memory an XML sub-tree that is no longer relevant. As long as the XML encoder/decoder is concerned, there is no consistency issue. If the generator is always required to generate valid document (fragments), then there is no need for a command to remove (possibly inconsistent or invalid) nodes or sub-trees. That is, only insert and append commands are needed.

A command packet contains the path of (the root of) the sub-tree to be appended or inserted and the packet number of the structure packet that contains the sub-tree. For example, returning to Fig. 6B, if the locator 608 for node B4 was not able to be accommodated in the packet 620, then a command packet would have to be inserted between packets #2 and #20 that effectively attaches node B4 to node A. That command packet would then include a locator pointing to the packet 612 including the structure defined by node B4.

The Definition of the Bitstream

The bitstream 306 is preferably defined in Extended Backus-Naur Form (ENBF) in the fashion defined by the Appendix. Characters are enclosed by single quote and strings by double quotes. Unless stated otherwise, UCS characters in UTF-8 encoding and UTF strings (that include length information) are assumed.

API

API for Documents and Schemas

It is not always necessary for the decoder 402 to convert an encoded document back into XML. As indicated above, the decoder 402 may support an API such as the SAX API, the DOM API, or other proprietary API, to allow an application to access the decoded content directly. This saves the decoder 402 from having to reconstruct and

output the XML document and the application from having to re-parse the reconstructed XML document.

An application may also have to access information stored in schemas. As schemas are also XML documents, they can be encoded in the same way. Using existing SAX or DOM API for accessing and interpreting schema definitions is extremely tedious. A parser that supports a schema API, such as the Schema API defined in Wan E., Anderson M., Lennon A., *Description Object Model (DesOM).* Doc. ISO/IEC JTC1/SC29/WG11 MPEG00/M5817, Noordwijkerhout, March 2000, will make accessing the definitions of schemas much easier.

To allow the values of built-in datatypes and special types to be encoded efficiently, an encoder has to be able to obtain type information from the schemas. Hence, a schema API is also extremely important to the encoder 302.

API for Encoders

The binary format proposed below allows for the implementation of encoders of various capabilities and complexity. The interfaces described in this section allow one to construct a basic encoder that can be extended to provide the more complicated features supported by the encoding scheme.

Encoder Interface

void SetMaxPacketSize(in unsigned long maxPacketSize)

- Set the maximum packet size in bytes.

void SetMaxPrivateDataSize(in unsigned long maxPrivateDataSize)

- Set the maximum size of the private data in byte. Note that the amount of private data that can be included in a packet is limited by the maximum size of the packet. A large

amount of private data is not expected as such works against the objective of reducing the size of the bitstream.

void SetHeaderUserData(in ByteArray headerData)

- Write the user data to the header packet. Any existing data will be overwritten.

void UseCodeTable(in CodeTable codeTable, in Boolean encodeIt)

- Inform the encoder of a pre-defined code table and whether the code table should be encoded with the data.

void SetCompressor(in Section section, in Inflater compressor)

- Instruct the encoder to use the specified compressor for the specified section. Section is an enumeration with the following values: STRUCT_BODY=1, TEXT_BODY=2, ID_TABLE=3, NS_SECT=4, ELEMENT_SECT=5, ATTR_NAME_SECT=6, ATTR_VALUE_SECT=7, ATTR_PAIR_SECT=8, DOC_HIERARCHY_SECT=9. Inflater has the same interface as Inflater of the java.util.zip package.

void Flush()

- Flush the packets in the buffer to the output stream.

void OnOutput()

- Receive notification before the set of packets in the buffer is output to allow the application to insert application specific-data to the packets.

void SetPacketUserData(in ByteArray userData)

- Write the user data to each of the packets except any header packet in the buffer. Any existing user data will be overwritten.

<u>Code Table Interface</u>

unsigned short GetSize()

- Get the number of entries in the code table.

wstring GetNamespace(in unsigned short i)

- Get the namespace of the value associated with the ith entry of the code table.

wstring GetValue(in unsigned short i)

- Get the value associated with the ith entry of the code table.

wstring GetType(in unsigned short i)

- Get the type of the value associated with the ith entry of the code table.

ByteArray GetCode(in unsigned short i)

- Get the code associated with the ith entry of the code table.

unsigned short GetIndexByCode(in ByteArray code)

- Get the value associated with a code.

unsigned short GetIndexByValue(in wstring value)

- Get the value associated with a code.

unsigned short GetMaxCodeValue()

- Get the maximum code value reserved by the code table. The encoder is free to use code value above the maximum code value. Depending on application, an encoder may also be implemented to use holes left by a pre-defined code table.

<u>Type Encoder Interface</u>

ByteArray Encode(in wstring text)

- Encode the value into a byte array given its text representation.

wstring Decode(in ByteArray encodedText)

- Decode an encoded value into the text representation of the value.

Encoding the XML Data, in particular MPEG-7 Descriptions of a Presentation

If (fragments of) XML data including MPEG-7 descriptions (which are XML data used for describing audio-visual (AV) content) are to be streamed and presented with AV content, the timing of and the sychronization between the media objects (including the

XML data) have to be specified. Like XML, the DDL (the description definition language of XML) does not define a timing and synchronization model for presenting media objects. As mentioned above, a SMIL-like MPEG-7 description scheme called herein Presentation Description Scheme is desired to provide the timing and synchronization model for authoring multimedia presentations.

It has been suggested that MPEG-7 descriptions can be treated in the same way as AV objects. This means that each MPEG-7 description fragment, like AV objects, used in a presentation will be tagged with a start time and a duration defining its temporal scope. This allows both MPEG-7 fragments and AV objects to be mapped to a class of media object elements of the Presentation Description Scheme and subjected to the same timing and sychronization model. Specifically, in the case of a SMIL-based Presentation Description Scheme, a new media object element such as an <mpeg7> tag can be defined. Alternately, MPEG-7 descriptions can also be treated as a specific type of *text*.

It is possible to send different types of MPEG-7 descriptions in a single stream or in separate streams. It is also possible to send an MPEG-7 description fragment that has sub-fragments of different temporal scopes in a single data stream or in separate streams. This is a role for the *presentation* encoder, in contrast to the XML encoder 300 discussed earlier.

The presentation encoder wraps an XML packet with a start time and a duration signalling when and for how long the content of the packet is required or relevant. The packet may contain:

(i) multiple short description fragments (each with their own temporal scope) concatenated together to achieve high compression rate and minimize overhead;

(ii) a single description fragment; and

(iii) part of a large description fragment.

In the case where the packet contains multiple description fragments, the start time of the packet is the earliest of the start times of the fragments while the duration of the packet is the difference between the latest of the end time of the fragments (calculated by adding the duration of the fragment to its start time) and the start time of the packet.

In broadcasting applications, to enable users to tune into the presentation at any time, relevant materials have to be repeated at regular interval. While only some of the XML packets have to be resent as some of the XML packets sent earlier may no longer be relevant, the header packet needs to be repeated. This means that, in the case of broadcasting applications, the header packet may be interspersed among structure, text and command packets to reset the transmission to a known state.

Industrial Applicability

It is apparent from the above that the arrangements described are applicable to the computer and data processing industries and to the efficient use of communication resources associated therewith whilst affording the ability to work with partially received information.

The foregoing describes only one or more embodiments of the present invention, and modifications and/or changes can be made thereto without departing from the scope and spirit of the invention, the embodiment(s) being illustrative and not restrictive. For example, whilst described with reference to XML documents, the procedures disclose herein are applicable to any hierarchical representation, such as a tree representation of a document.

Appendix:

Definition of the Bitstream

The bitstream will be defined in Extended Backus-Naur Form (ENBF). Character will be enclosed by single quote and string by double quote. Unless stated otherwise, UCS characters in UTF-8 encoding and UTF strings (that include length information) are assumed.

xmlbitStream ::= xml_packet+

N.B.: The bitstream of an encoded XML document consists of a sequence of packets. The sequence begins with a header packet and ends with a trailer packet.

Packet

xml_packet ::= packet_header packet_body

packet_header ::= packet_signature packet_number packet_size packet_type packet_private_data

packet_number ::= variable_length_natural_number

N.B.: packet_number has to be greater than 0.

packet_type ::= header_packet | structure_packet | text_packet | trailer_packet | command_packet

packet_signature ::= 'x' 'm' 'l' 'b' 'i' 'n' 'p' 'k'

packet_size ::= unsigned_short

N.B.: With unsigned_short, an unsigned integer in the range 0 - 65535 is represented using 2 bytes with the first byte being the hihg-order byte of the integer.

packet_private_data ::= byte_array

packet_body ::= header_body | trailer_body | structure_body | text_body | command_body

header_packet ::= 'h'

structure_packet ::= 's'

text_packet ::= 't'
trailer_packet ::= 'e'
command_packet ::= 'c'
byte_array ::= size_in_byte byte*
size_in_byte ::= variable_length_natural_number

N.B.: With variable_length_natural_number, a natural number in the range 0 - 1,073,741,823 is represented using 1 to 4 bytes with the first byte being the high-order byte of the number. The two most significant bits of the high-order byte is actually used to indicate the number of additional bytes used for representing the number. For instance, '01' implies one additional byte or a 2-byte representation and '11' implies 3 additional bytes or a 4-byte representation.)

Byte ::= [#x00 #xFF]

Header

header_body ::= enoding_version xml_version xml_params max_packet_size max_decompressed_packet_size max_packet_number section_compressor_list type_encoder_list
xml_params ::= count xml_encoding xml_standalone
encoding_version ::= "1.0"
xml_version ::= "1.0"
count ::= variable_length_natural_number
xml_encoding ::= UTF8_string

N.B.: With UTF8_String, the first two bytes is an unsigned short, the *UTF length*, that specifies the number of additional bytes to be read. The additional bytes contain the UTF-8 encoding of the string.

xml_standalone ::= 'y' | 'n'
max_packet_size ::= variable_length_natural_number

N.B.: A value of zero implies that the maximum packet size is unknown.

max_packet_number ::= variable_length_natural_number

N.B.: A value of zero implies that the maximum number of packets is unknown.

section_compressor_list ::= count (section_ID compressor_URI)*
type_encoder_list ::= count (type_ID type_encoder_URI)*
compressor_URI ::= URI

```
type_encoder_URI ::= URI
URI              ::= UTF8_string
section_ID       ::= struct_body_ID | text_body_ID | id_table_ID | ns_section_ID |
                     element_sect_ID | attribute_name_sect_ID | attribute_value_sect_ID
                     | attribute_pair_sect_ID | doc_hierarchy_sect_ID

struct_body_ID   ::= 's'
text_body_ID     ::= 't'
id_table_ID      ::= 'i'
ns_section_ID    ::= 'n'
element_sect_ID  ::= 'e'
attribute_name_sect_ID      ::= 'a'
attribute_value_sect_ID     ::= 'v'
attribute_pair_sect_ID      ::= 'p'
doc_hierarchy_sect_ID       ::= 'd'
type_ID          ::= [ #x00    #xFF ]
other_type_ID    ::= #x00
string_ ID       ::= #x01
string_locator_ID ::= #x02
boolean_ID       ::= #x03
byte_ID          ::= #x04
unsigned_short_ID ::= #x05
short_ID         ::= #x06
unsigned_long_ID ::= #x07
long_ID          ::= #x08
float_ID         ::= #x09
double_ID        ::= #x0A
date_ID          ::= #x0B
time_ID          ::= #x0C
```

N.B.: The above list for built-in datatypes are not complete. Type 00-0F are for built-in datatypes. An XML encoder can assign type 10-FF to application-specific types. The application is responsible for providing the (Java) type encoder and decoder for any application-specific types. These type encoder and decoder must be pre-installed or downloaded before they are required. When type information is

not available, XML text and attribute values will be treated as string.

Trailer

trailer_body ::=

N.B.: At the moment, the trailer packet is only used to signal the end of the XML document. The body of the trailer packet is empty.

Structure Packet

structure_body ::= [ID_table_section] [internal_subset_section] [ns_table_section] [element_name_codetable_section] [attribute_name_codetable_section] [attribute_value_codetable_section] [attribute_name_value_pair_section] [document_hierarchy_section]

N.B.: Although the above EBNF rule defines the various sections of the body of a structure packet to be arranged in a particular order, the sections are actually allowed to be arranged in any order as each section is identified by its unique signature.

ID Table Section

ID_table_section ::= ID_table_section_signature section_size compressed entry_count (ID_table | compressed_ID_table)

section_size ::= size_in_byte

N.B.: section_size stores the size of the section excluding its signature.

compressed ::= boolean

N.B.: The compressed flag indicates whether the table is compressed.

N.B.: With boolean, a byte value of 1 represents true an a byte value of 0 represents false.

entry_count ::= variable_length_natural_number

size_of_compressed_ID_table ::= variable_length_natural_number

ID_table ::= (ID_string offset_to_the_document_hierarchy)*

N.B.: ID_table defines the structure of the uncompressed ID table. The ID table only collects ID of nodes (not including nodes referred to by node locators) that appears in the document hierarchy of the same packet. If type information is not available during encoding, IDs will

not be collected into the ID table even if they are present in the document as there is no way the encoder can identify them.

ID_string ::= UTF8_string

offset_to_the_document_hierarchy ::= byte_offset

N.B.: offset_to_the_document_hierarchy is the byte offset to document_hierarchy in the (uncompressed) document_hierarchy_section not the byte offset to the (uncompressed) document_hierarhy_section

byte_offset ::= variable_length_natural_number

ID_table_section_signature ::= #xFF01

Internal Subset Section

Internal_subset_section ::= internal_subset_section_signature section_size compressed [byte*]

N.B.: The detail of the internal subset section has yet to be defined.

NS_table_section_signature ::= #xFF02

Namespace Table Section

NS_table_section ::= NS_table_section_signature section_size compressed entry_count index_base (NS_table | compressed_NS_table)

index_base ::= variable_length_natural_number

N.B.: The index into the NS_table is used as the namespace code. The base of the index is specified in the field index_base. The namespace code 0 is reserved for the null namespace. Hence, a namespace table cannot have an index_base of 0.

NS_table ::= (NS_URI)*

N.B.: NS_table defines the structure of the uncompressed NS table. The index into the table is used as the namespace code. The base of the index is specified in the field index_base. The namespace code 0 is reserved for the null namespace. Hence, a namespace table cannot have an index_base of 0.

NS_URI ::= URI

NS_table_section_signature ::= #xFF03

Code Table Sections

element_name_codetable_section ::= element_name_codetable_section_signature section_size compressed entry_count index_base (element_name_codetable | compressed_element_name_codetable)

attribute_name_codetable_section ::= attribute_name_codetable_section_signature section_size compressed entry_count index_base (attribute_name_codetable | compressed_attribute_name_codetable)

attribute_value_codetable_section ::= attribute_value_codetable_section_signature section_size compressed entry_count index_base has_predefined_code (attribute_value_codetable | compressed_attribute_value_codetable)

N.B.: The index into each code table is used as the code unless there is a predefined code. The code tables allow the mapping between the codes used for the encoding and the actual values. The base of the index for each table is specified in the field index_base of that table. Only positive codes are allowed. Hence, index_base cannot have a value of zero.

element_name_codetable_section_signature ::= #xFF04

attribute_name_codetable_section_signature ::= #xFF05

attribute_value_codetable_section_signature ::= #xFF06

has_predefined_code ::= boolean

N.B.: The has_predefined_code flag specify whether the code table has a predefined_code column.

Element name code table

element_name_codetable ::= element_name_code_table_entry*

N.B.: element_name_codetable defines the structure of the uncompressed element name code table. The index into the table is used as the element name code unless there is a predefined code. The base of the index is specified in the field index_base. The code 0 is reserved. Hence, a code table cannot have an index_base of 0.

element_name_codetable_entry ::= ns_code element_name type_ID [predefined_code]

N.B.: Except for the built-in datatypes and special types that are known to the encoder, textual content of all other type will be encoded as string.

predefined_code ::= byte_array

N.B.: An empty predefined_code implies that there is no predefined code for that entry. This should not happen. If an value is missing from a pre-defined code table. The encoder has to generate a code for the value and store it in the predefined_code field.

element_name ::= non_empty_UTF8_string | (#x0000 string_locator)

N.B.: The element names are usually stored in-line in the table. However, if an element name is too long, it can be stored in a separate text packet and a string locator is used in the table instead.

string_locator ::= text_packet_number byte_offset

N.B.: A byte_offset specifies the offset into the text packet's body where the string can be found.

non_empty_UTF8_string ::= UTF8_string – ""

Attribute name code table

attribute_name_codetable ::= attribute_name_code_table_entry*

N.B.: attribute_name_codetable defines the structure of the uncompressed attribute name code table. The index into the table is used as the attribute name code unless there is a predefined code. The base of the index is specified in the field index_base. The code 0 is reserved. Hence, a code table cannot have an index_base of 0.

attribute_name_codetable_entry ::= ns_code attribute_name type_ID [predefined_code]

N.B.: Except for the built-in datatypes and special types that are known to the encoder, textual content of all other type will be encoded as string.

attribute_name ::= non_empty_UTF8_string | (#x0000 string_locator)

N.B.: The attribute names are usually stored in-line in the table. However, if an attribute name is too long, it can be stored in a separate text packet and a string locator is used in the table instead.

Attribute value code table

attribute_value_codetable ::= attribute_value_code_table_entry*

N.B.: attribute_value_codetable defines the structure of the uncompressed attribute value code table. The index into the table is used as the

attribute value code unless there is a predefined code. The base of the index is specified in the field index_base. The code 0 is reserved. Hence, a code table cannot have an index_base of 0.

attribute_value_codetable_entry ::= ns_code attribute_value type_ID [predefined_code]

N.B.: Except for the built-in datatypes and special types that are known to the encoder, textual content of all other type will be encoded as string.

attribute_value ::= encoded_value

N.B.: The attribute value are usually stored in-line in the table.

encoded_value ::= encoded_value_of_non_string_type | non_empty_UTF8_string | ("" #x00) | (#x0000 string_locator)

N.B.: Values are encoded according to their types. Except for built-in datatypes and special types that are known to the encoder, value are encoded as string.

N.B.: An empty UTF8-string has to be followed by #x00 to distinguish it from a valid string locator. Again, if an attribute name is too long, it can be stored in a separate text packet and a string locator is used in the table instead.

Attribute Name/Value Pair Section

attribute_name_value_pair_section ::= attribute_name_value_pair_section_signature section_size compressed entry_count index_base (attribute_name_value_pair_table | compressed_attribute_name_value_pair_table)

attribute_name_value_pair_table = attribute_name_value_pair_entry*

N.B.: attribute_name_value_pair_table defines the structure of the uncompressed attribute name/value pair table. The base of the index (> 0) is specified in the field index_base.

attribute_name_value_pair_entry ::= attribute_name_code attribute_value_code

attribute_name_value_pair_section_signature ::= #xFF07

Document Hierarchy Section

document_hierarchy_section::= document_hierarchy_section_signature section_size compressed (subtree | compressed_subtree)

subtree ::= node

N.B.: subtree defines the structure of the uncompressed XML sub-tree.

node ::= node_size node_type (element_node | text_node | comment_node | node_locator)

N.B.: The node_size includes the size of the node and its descendent nodes encoded in the same packet.

node_type ::= (element_node_signature element_flag) | ((text_node_signature | comment_node_signature | locator_node_signature) #x0)

element_node_signature ::= #x3

text_node_signature ::= #x5

comment_node_signature ::= #x9

locatore_node_signature ::= #xC

element_flag ::= has_attributes | has_children | has_attributes_and_children

has_attributes ::= 0x1

has_children ::= 0x2

has_attributes_and_children ::= 0x3

element_node ::= element_name_code [attributes] [child_node*]

child_nodes ::= node

attributes ::= index_of_starting_attribute_name_value_pair number_of_attributes

number_of_attributes ::= variable_length_natural_number

text_locator ::= string_locator

comment_node ::= text_locator

node_locator ::= packet_number

Text Packet

text_body ::= compressed (encoded_text | compressed_encoded_text)

encoded_text ::= (0x00 encoded_value*) | (next_packet_number UTF8_string)

N.B.: If next_packet_number is zero, the first string of the text packet may be the last fragments of a long string. If next_packet_number is non-zero, the whole text packet contains a single fragment of a string.

next_packet_number ::= variable_length_natural_number

Command Packet

command_body ::= command path packet_number_of_subtree

N.B.: The subtree to be added is defined in the structure packet with the specified packet number.

command ::= insert_command | append_command

insert_command ::= #x01

append_command ::= #x02

path ::= URI_reference

URI-reference ::= UTF8_string

End of Appendix

CLAIMS:

1. A method of communicating at least part of a structure of a document described by a hierarchical representation, said method comprising the steps of:

identifying said representation of said document;

packetizing said representation into a plurality of data packets, said packets having a predetermined size, said packetizing comprising creating at least one link between a pair of said packets, said link representing an interconnection between corresponding components of said representation; and

forming said data packets into a stream for communication wherein said links maintain said representation within said packets.

2. A method according to claim 1 further comprising the steps of:

receiving said stream;

decoding said packets from said stream to identify said links;

using said links to reconstruct said representation for those portions of said representation not packetized with one packet of said stream.

3. A method according to claim 1 or 2 wherein said corresponding components comprise at least one structure component and a one content component of said document.

4. A method of communicating at least part of the structure of a document described by a hierarchical representation, said method comprising the steps of:

identifying at least one part of said representation and packetizing said parts into at least one packet of predetermined size; and

where any one or more of said parts of said representation do not fit within one said packet, defining at least one link from said one packet to at least one further said packet into which said non-fitting parts are packetized, said link maintaining the hierarchical structure of said document in said packets.

5. A method according to any one of claims 1 to 4 wherein said hierarchical representation comprises a tree representation.

6. A method according to any one of claims 1 to 5 wherein said document comprises an XML document.

7. A method according to any one of claims 1 to 6 wherein said predetermined size is a predetermined maximum size.

8. A method of facilitating access to the structure of an XML document, said method comprising the steps of:

identifying a hierarchical representation of said document;

packetizing said representation into a plurality of packets of predetermined packet size;

forming links between said packets to define those parts of said representation not able to be expressed within a packet thereby enabling reconstruction of said representations after de-packetizing.

9. A method of encoding an XML document, said method comprising the steps of:

examining said XML document to identify each data type forming part of said XML document;

identifying a first set of said data types for which a corresponding special encoding format is available;

first encoding each part of said XML document having a data type in said first set with the corresponding special encoding format;

second encoding each remaining part of said XML document with a default encoding format corresponding to the data type of said remaining part;

forming a representation of information referencing at least each said data type in said first set with the corresponding special encoding format; and

associating said representation and said encoded parts as an encoded form of said XML document.

10. A method according to claim 9 wherein said encoding separately encodes structure and content parts of said XML document, and said representation is retained in a header portion of the encoded form of said XML document.

11. A method according to claim 10 wherein said representation is retained in said header portion as a table.

12. A method according to claim 10 wherein said separately encoded parts comprise packets of at least one bitstream.

13. A method according to claim 9 wherein said first encoding comprises examining said special data type and said corresponding part and determining a one of said encoding formats to be applied to said part.

14. A method according to claim 9 wherein said encoding comprises selecting one of a plurality of said encoding formats corresponding to a data type of said part and encoding said part with said selected encoding format.

15. A method of decoding an encoded XML document, said method comprising the steps of :

examining said encoded XML document to identify an encoding format associated with each data type forming part of said XML document; and

decoding each said part using a decoder complementing the encoding format with which the corresponding data type was encoded.

16. A method according to claim 15 wherein said encoded XML document comprises separately encoded structure parts and content parts of said XML document and said examining comprises identifying from within at least one structure part a representation associating an encoded part of said XML document with the corresponding encoding format.

17. A method according to claim 16 wherein said encoded XML document is formed as a plurality of packets and said representation is formed within a header of at least one said packet.

18. A method according to claim 17 wherein said representation comprises a table.

19. A method according to any one of claims 16 to 18 wherein said representation comprises a URI.

20. A method according to claim 16 wherein said representation relates to at least a set of special data types having corresponding decoding formats

21. A method according to claim 15 wherein if a decoder is not available to decode a part of said encoded XML document, said method comprises ignoring said part.

22. A packetized bitstream formed using the method of any one of claims 1 to 14.

23. Apparatus for communicating at least part of a structure of a document described by a hierarchical representation, said apparatus comprising:

means for identifying said representation of said document;

means for packetizing said representation into a plurality of data packets, said packets having a predetermined size, said packetizing comprising creating at least one link between a pair of said packets, said link representing an interconnection between corresponding components of said representation; and

means for forming said data packets into a stream for communication wherein said links maintain said representation within said packets.

24. Apparatus for encoding an XML document, said apparatus comprising:

means for examining said XML document to identify each data type forming part of said XML document;

means for identifying a first set of said data types for which a corresponding special encoding format is available;

means for first encoding each part of said XML document having a data type in said first set with the corresponding special encoding format;

means for second encoding each remaining part of said XML document with a default encoding format corresponding to the data type of said remaining part;

means for forming a representation of information referencing at least each said data type in said first set with the corresponding special encoding format; and

means for associating said representation and said encoded parts as an encoded form of said XML document.

25. A computer readable medium, having a program recorded thereon, where the program is configured to make a computer execute a procedure for communicating at least part of a structure of a document described by a hierarchical representation, said program comprising steps for:

identifying said representation of said document;

packetizing said representation into a plurality of data packets, said packets having a predetermined size, said packetizing comprising creating at least one link between a pair of said packets, said link representing an interconnection between corresponding components of said representation; and

forming said data packets into a stream for communication wherein said links maintain said representation within said packets.

26. A computer readable medium according to claim 25 further comprising the steps for:

receiving said stream;

decoding said packets from said stream to identify said links;

using said links to reconstruct said representation for those portions of said representation not packetized with one packet of said stream.

27. A computer readable medium according to claim 25 or 26 wherein said corresponding components comprise at least one structure component and a one content component of said document.

28. A computer readable medium, having a program recorded thereon, where the program is configured to make a computer execute a procedure for communicating at least part of the structure of a document described by a hierarchical representation, said program comprising steps for:

identifying at least one part of said representation and packetizing said parts into at least one packet of predetermined size; and

where any one or more of said parts of said representation do not fit within one said packet, defining at least one link from said one packet to at least one further said packet into which said non-fitting parts are packetized, said link maintaining the hierarchical structure of said document in said packets.

29. A computer readable medium according to any one of claims 25 to 28 wherein said hierarchical representation comprises a tree representation.

30. A computer readable medium according to any one of claims 25 to 29 wherein said document comprises an XML document.

31. A computer readable medium according to any one of claims 25 to 30 wherein said predetermined size is a predetermined maximum size.

32. A computer readable medium, having a program recorded thereon, where the program is configured to make a computer execute a procedure to facilitating access to the structure of an XML document, said program comprising steps for:

identifying a hierarchical representation of said document;

packetizing said representation into a plurality of packets of predetermined packet size;

forming links between said packets to define those parts of said representation not able to be expressed within a packet thereby enabling reconstruction of said representations after de-packetizing.

33. A computer readable medium, having a program recorded thereon, where the program is configured to make a computer execute a procedure to encode an XML document, said program comprising steps for :

examining said XML document to identify each data type forming part of said XML document;

identifying a first set of said data types for which a corresponding special encoding format is available;

first encoding each part of said XML document having a data type in said first set with the corresponding special encoding format;

second encoding each remaining part of said XML document with a default encoding format corresponding to the data type of said remaining part;

forming a representation of information referencing at least each said data type in said first set with the corresponding special encoding format; and

associating said representation and said encoded parts as an encoded form of said XML document.

34. A computer readable medium according to claim 33 wherein said encoding separately encodes structure and content parts of said XML document, and said representation is retained in a header portion of the encoded form of said XML document.

35. A computer readable medium according to claim 34 wherein said representation is retained in said header portion as a table.

36. A computer readable medium according to claim 34 wherein said separately encoded parts comprise packets of at least one bitstream.

37. A computer readable medium according to claim 33 wherein said first encoding comprises examining said special data type and said corresponding part and determining a one of said encoding formats to be applied to said part.

38. A computer readable medium according to claim 33 wherein said encoding comprises selecting one of a plurality of said encoding formats corresponding to a data type of said part and encoding said part with said selected encoding format.

39. A computer readable medium, having a program recorded thereon, where the program is configured to make a computer execute a procedure to decode an encoded XML document, said program comprising steps for:

examining said encoded XML document to identify an encoding format associated with each data type forming part of said XML document; and

decoding each said part using a decoder complementing the encoding format with which the corresponding data type was encoded.

40. A computer readable medium according to claim 39 wherein said encoded XML document comprises separately encoded structure parts and content parts of said XML document and said examining comprises identifying from within at least one structure part a representation associating an encoded part of said XML document with the corresponding encoding format.

41. A computer readable medium according to claim 40 wherein said encoded XML document is formed as a plurality of packets and said representation is formed within a header of at least one said packet.

42. A computer readable medium according to claim 41 wherein said representation comprises a table.

43. A computer readable medium according to any one of claims 40 to 42 wherein said representation comprises a URI.

44. A computer readable medium according to claim 43 wherein said representation relates to at least a set of special data types having corresponding decoding formats

45. A computer readable medium according to claim 44 wherein if a decoder is not available to decode a part of said encoded XML document, said method comprises ignoring said part.

46. Apparatus for communicating at least part of a structure of a document described by a hierarchical representation, said apparatus comprising:

an identifying unit which identifies said representation of said document;

a packetizing unit which packetizes said representation into a plurality of data packets, said packets having a predetermined size, said packetizing comprising creating at least one link between a pair of said packets, said link representing an interconnection between corresponding components of said representation; and

a forming unit which forms said data packets into a stream for communication wherein said links maintain said representation within said packets.

47. Apparatus for encoding an XML document, said apparatus comprising:

an examining unit which examines said XML document to identify each data type forming part of said XML document;

an identifying unit which identifies a first set of said data types for which a corresponding special encoding format is available;

a first encoding unit which encodes each part of said XML document having a data type in said first set with the corresponding special encoding format;

a second encoding unit which encodes each remaining part of said XML document with a default encoding format corresponding to the data type of said remaining part;

a forming unit which forms a representation of information referencing at least each said data type in said first set with the corresponding special encoding format; and

an associating unit which associates said representation and said encoded parts as an encoded form of said XML document.

48. A method of encoding a document described by a hierarchical representation substantially as described herein with reference to the drawings.

49. A method of decoding a packetized stream incorporating a hierarchical representation substantially as described herein with reference to the drawings.

50. A method of communicating a document described using a hierarchical representation substantially as described herein with reference to the drawings.

51. Apparatus for performing the method of any one of claims 48 to 50.

52. A computer readable medium having a computer program recorded thereon for performing the method of any one of claims 48 to 50.

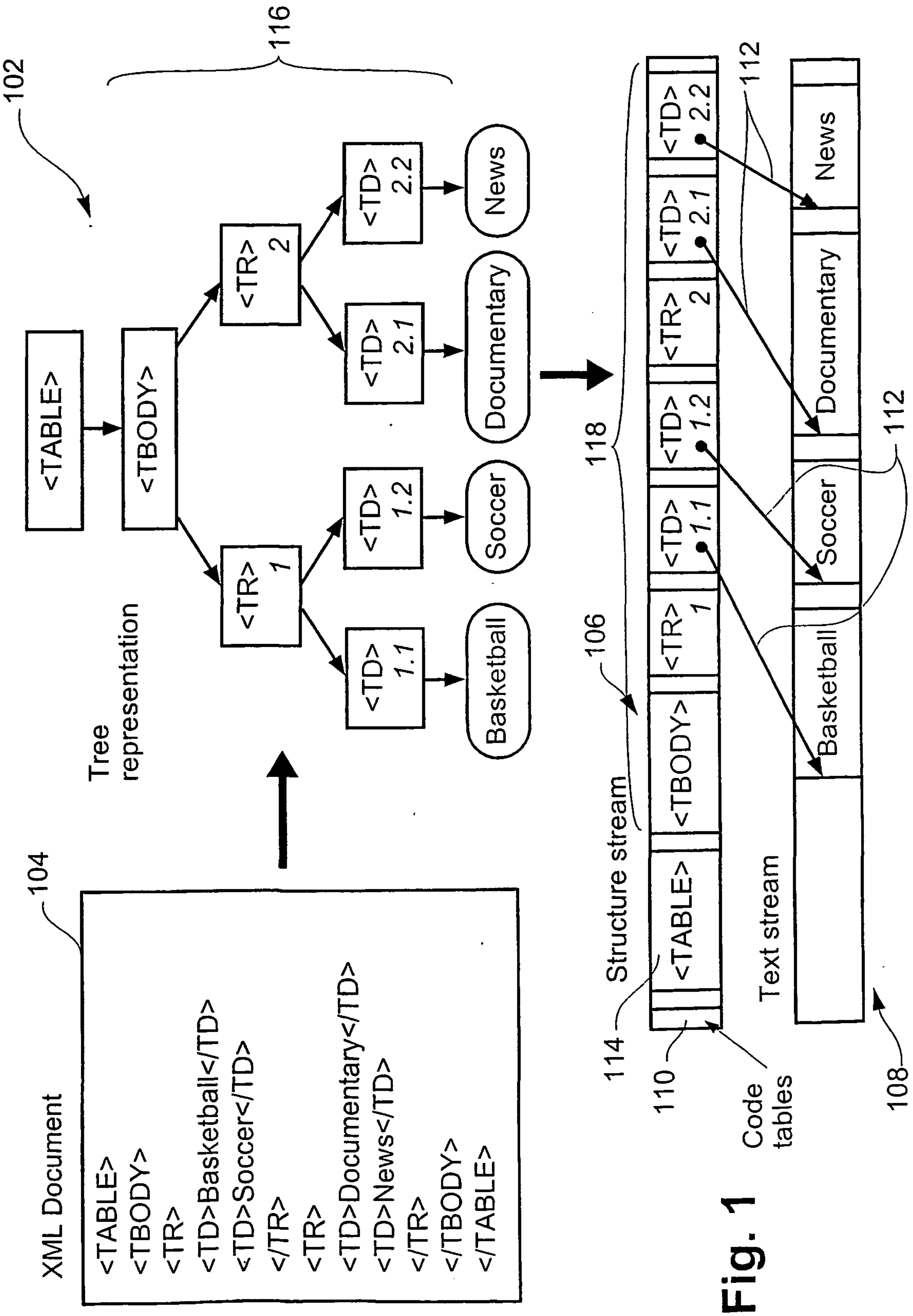

XML Document
<TABLE>
<TBODY>
<TR>
<TD>Basketball</TD>
<TD>Soccer</TD>
</TR>
<TR>
<TD>Documentary</TD>
<TD>News</TD>
</TR>
</TBODY>
</TABLE>
104
Tree representation
102
116
<TABLE>
<TBODY>
<TR> 1
<TR> 2
<TD> 1.1
<TD> 1.2
<TD> 2.1
<TD> 2.2
Basketball
Soccer
Documentary
News
Structure stream
106
118
112
114
110
Code tables
Text stream
108

Fig. 1

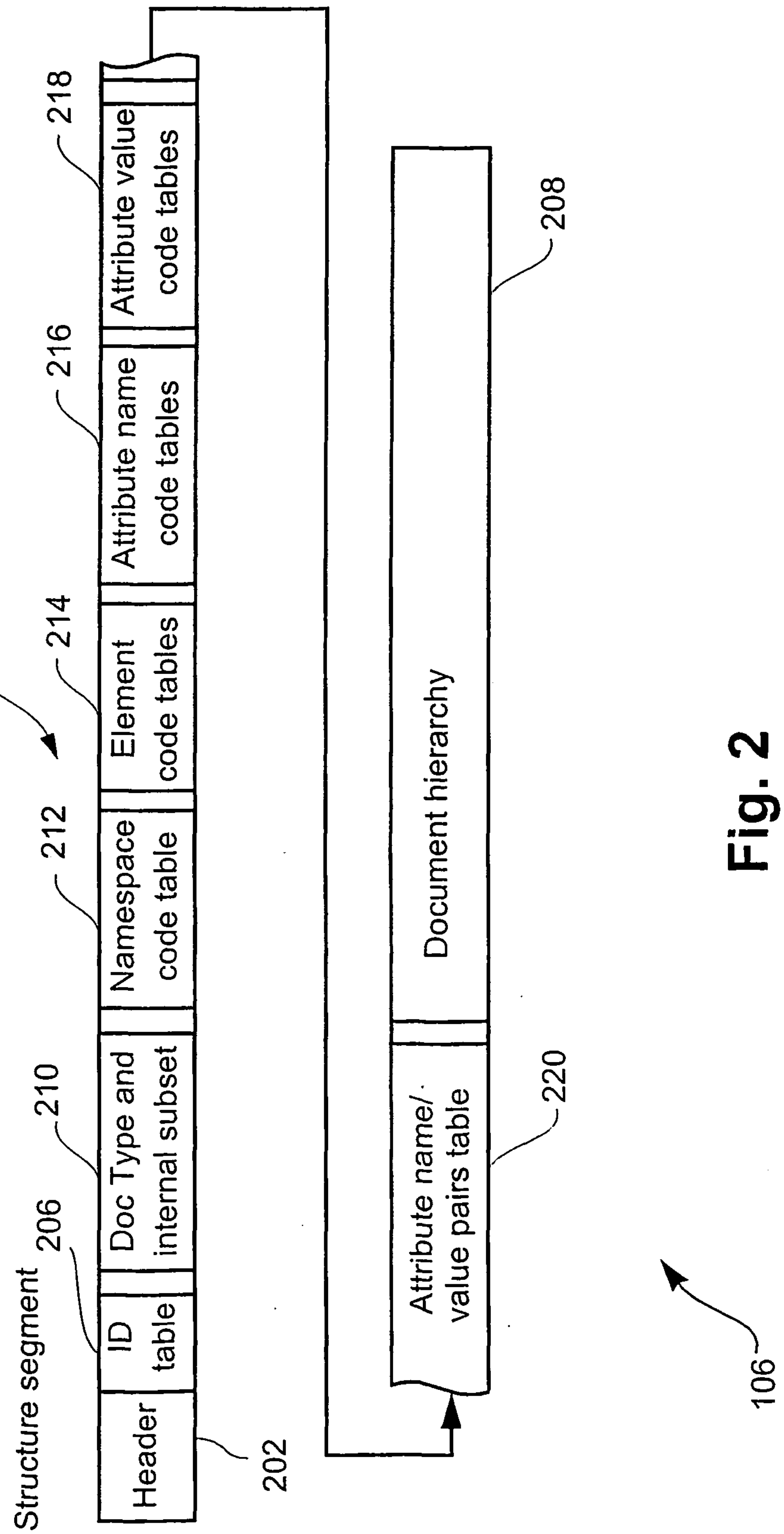

Structure segment
Header
202
ID table
206
Doc Type and internal subset
210
Namespace code table
212
Element code tables
214
204
Attribute name code tables
216
Attribute value code tables
218
Attribute name/ value pairs table
220
Document hierarchy
208
106

Fig. 2

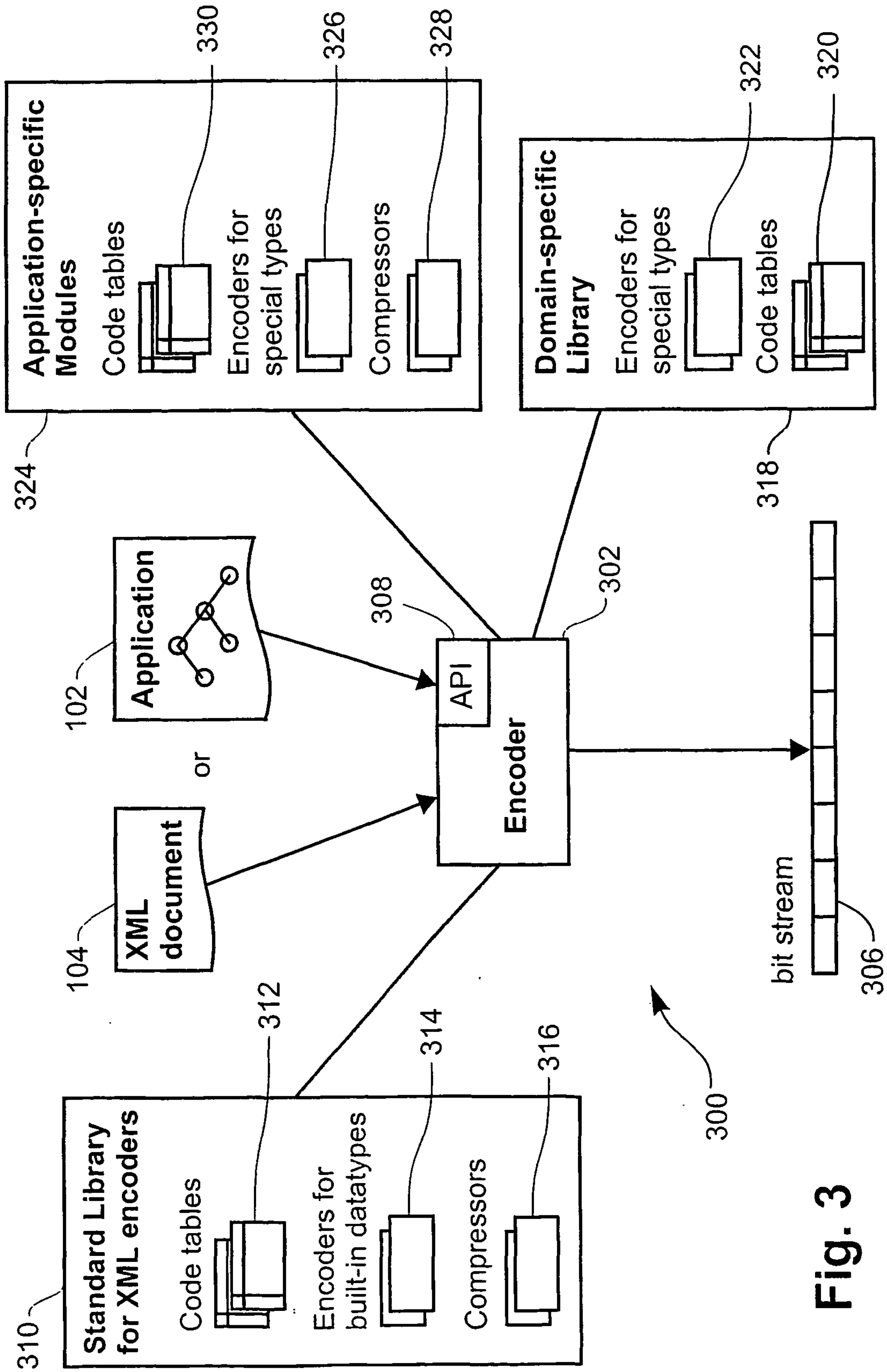

Application-specific Modules
Code tables
Encoders for special types
Compressors
330
326
328
324
Domain-specific Library
Encoders for special types
Code tables
322
320
318
Application
102
or
XML document
104
308
API
Encoder
302
bit stream
306
300
Standard Library for XML encoders
Code tables
Encoders for built-in datatypes
Compressors
312
314
316
310

Fig. 3

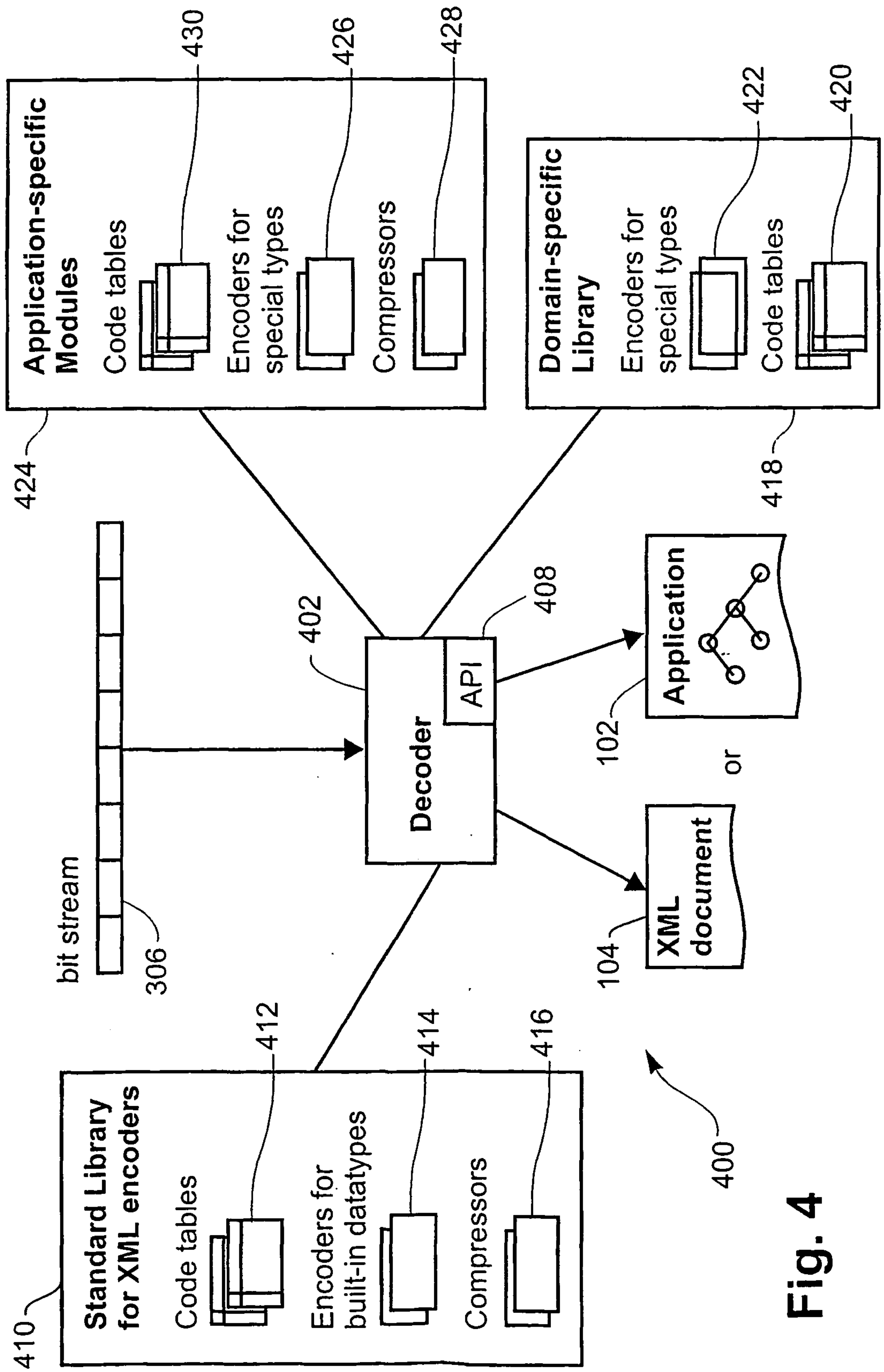

Application-specific Modules
Code tables
Encoders for special types
Compressors
430
426
428
424
Domain-specific Library
Encoders for special types
Code tables
422
420
418
bit stream
306
Decoder
API
402
408
Application
102
or
XML document
104
Standard Library for XML encoders
Code tables
Encoders for built-in datatypes
Compressors
410
412
414
416
400

Fig. 4

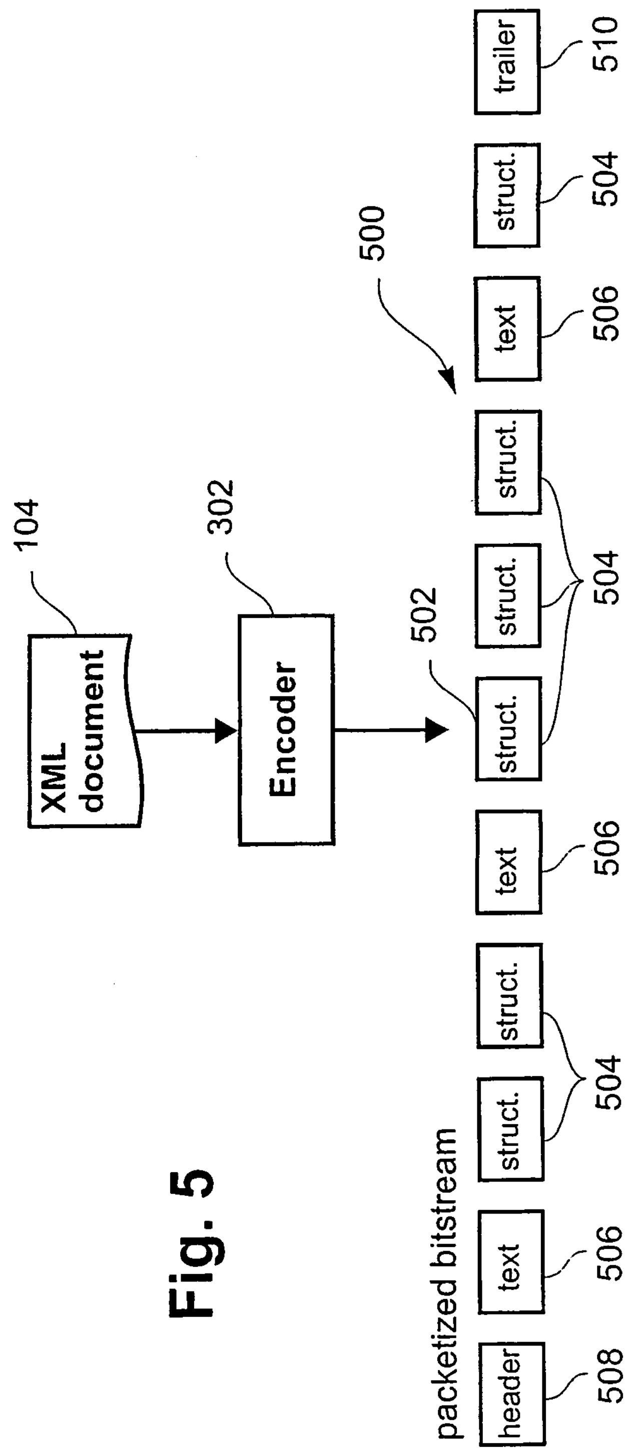

XML document
104
Encoder
302
packetized bitstream
header
text
struct.
struct.
struct.
text
struct.
struct.
struct.
text
struct.
trailer
508
506
504
506
502
504
500
506
504
510

Fig. 5

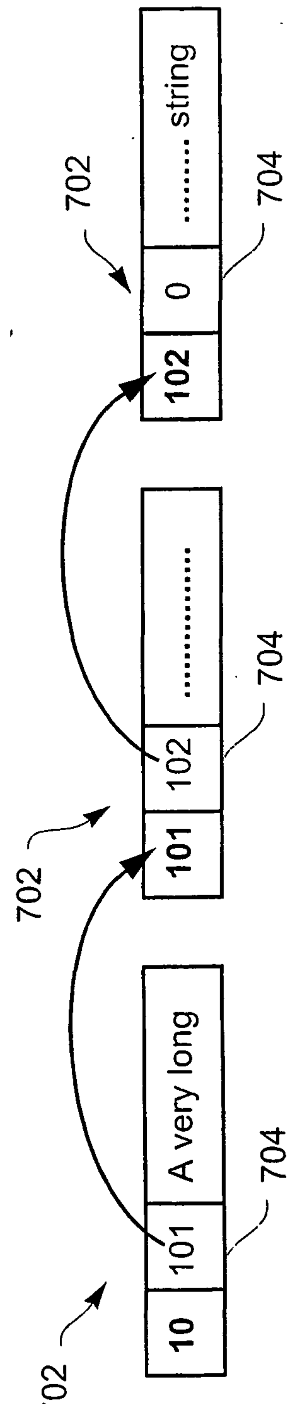

10
101
A very long
101
102
....
102
0
.... string
702
702
702
704
704
704

Fig. 7

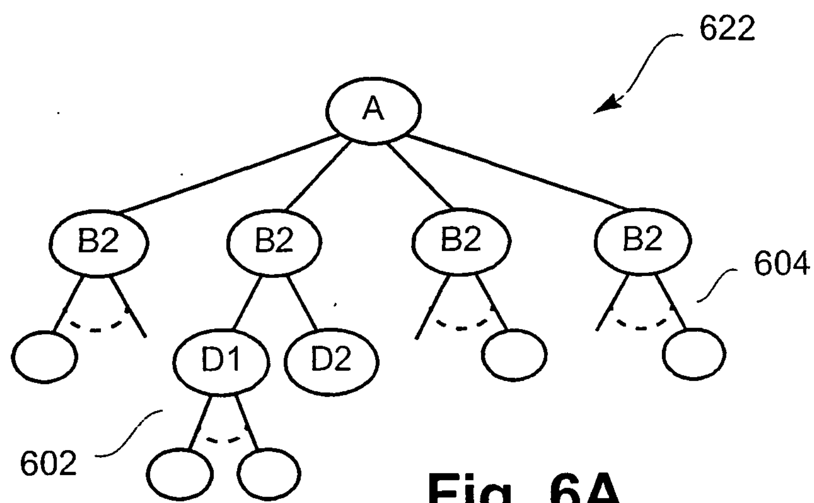

622
A
B2
B2
B2
B2
604
D1
D2
602

Fig. 6A

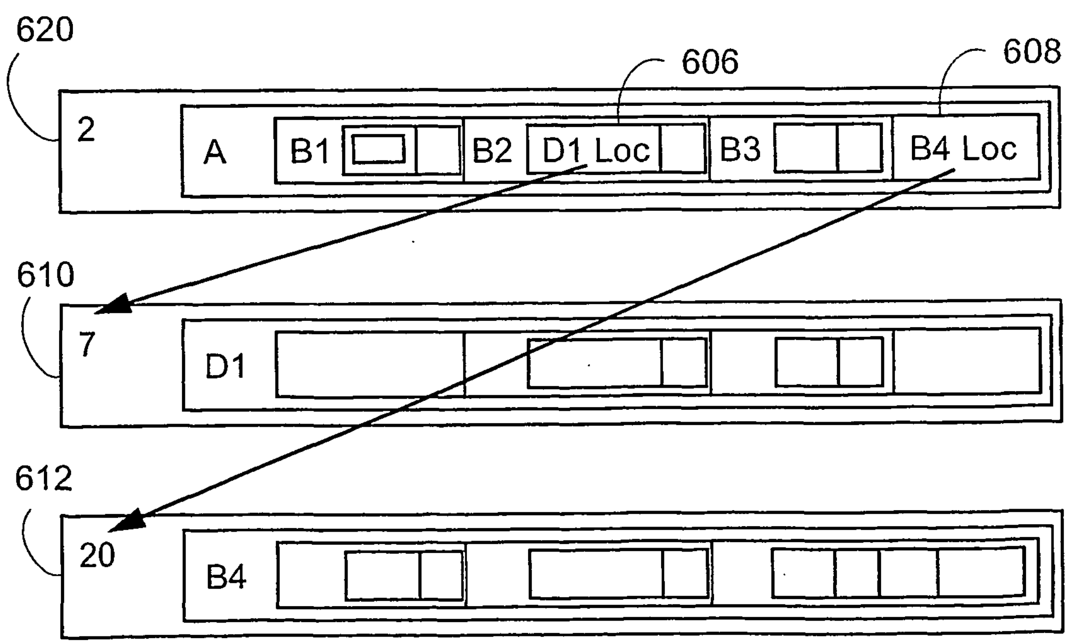

620
606
608
2
A
B1
B2
D1 Loc
B3
B4 Loc
610
7
D1
612
20
B4

Fig. 6B

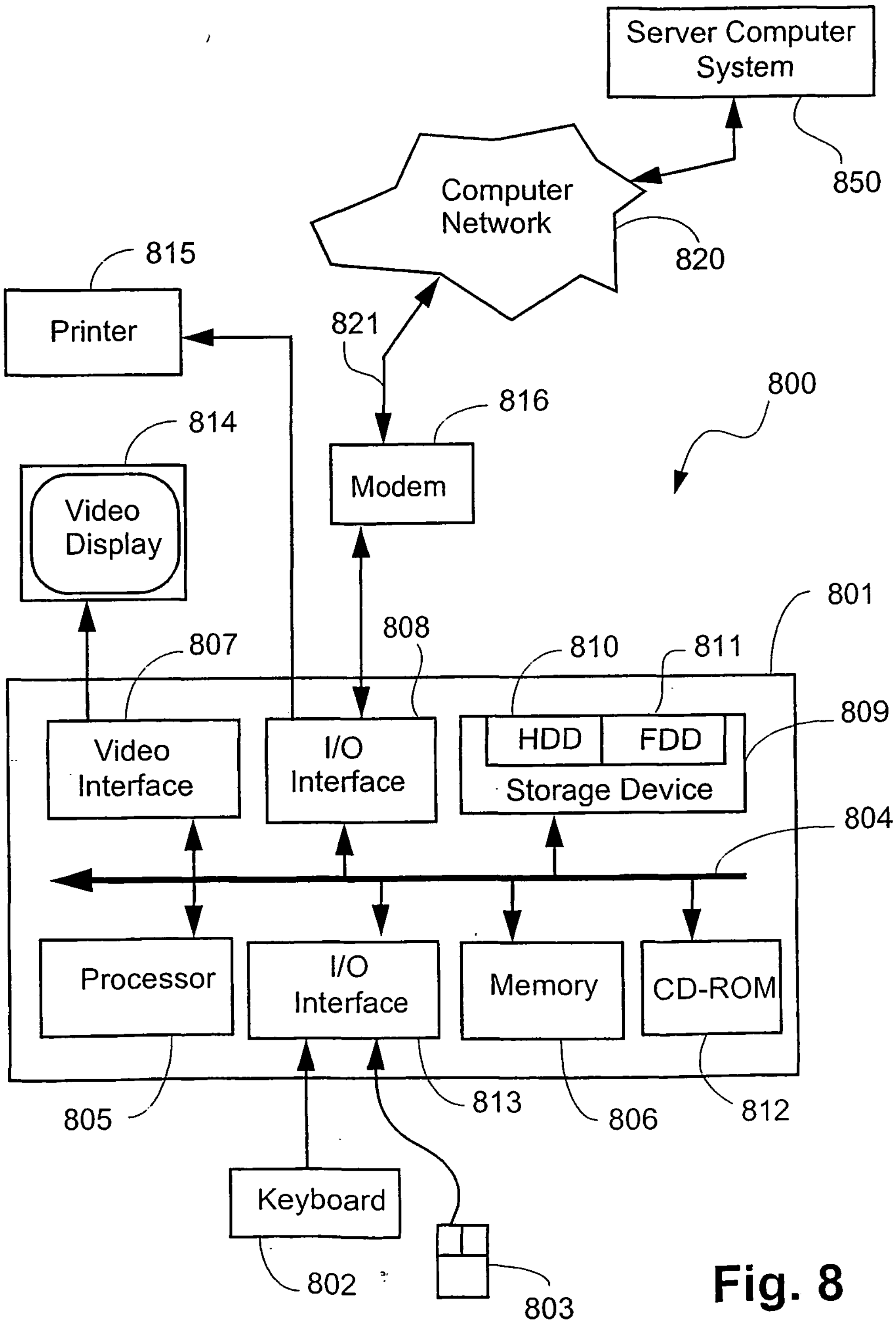

Server Computer System
850
Computer Network
820
815
Printer
821
816
800
814
Video Display
Modem
801
807
808
810
811
809
Video Interface
I/O Interface
HDD
FDD
Storage Device
804
Processor
I/O Interface
Memory
CD-ROM
805
813
806
812
Keyboard
802
803

Fig. 8

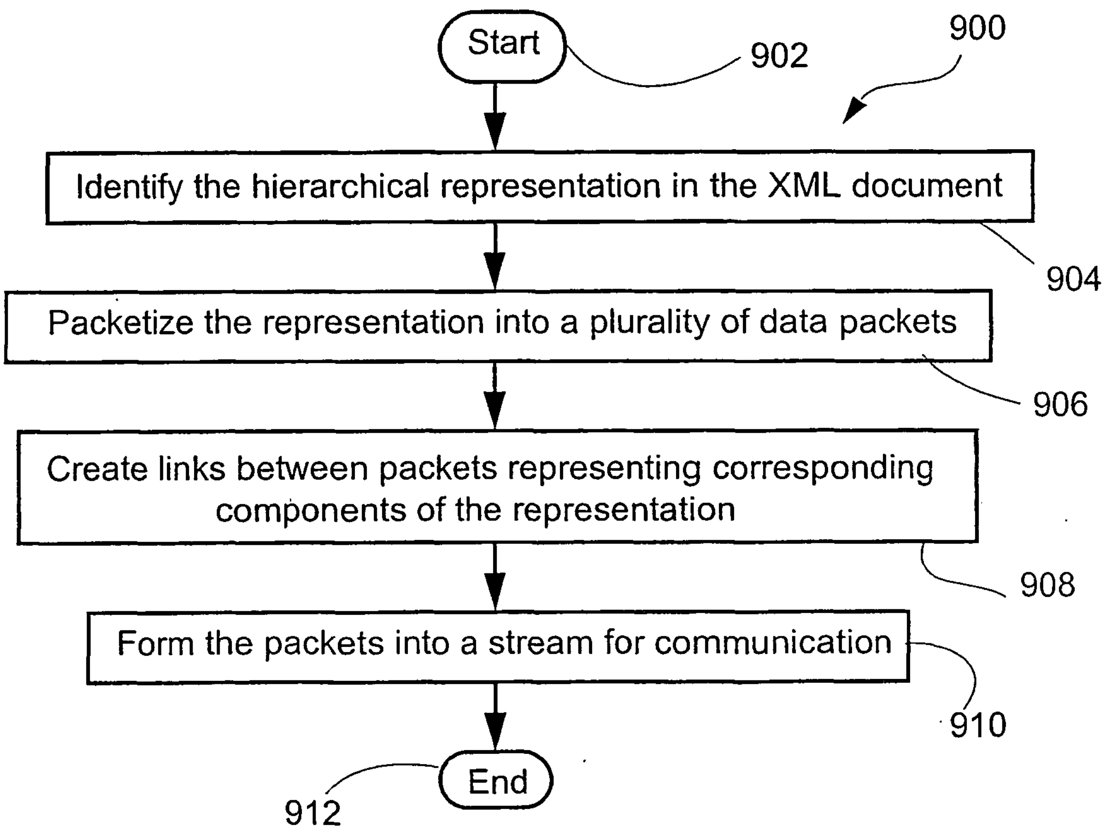

Start
902
900
Identify the hierarchical representation in the XML document
904
Packetize the representation into a plurality of data packets
906
Create links between packets representing corresponding components of the representation
908
Form the packets into a stream for communication
910
End
912

Fig. 9

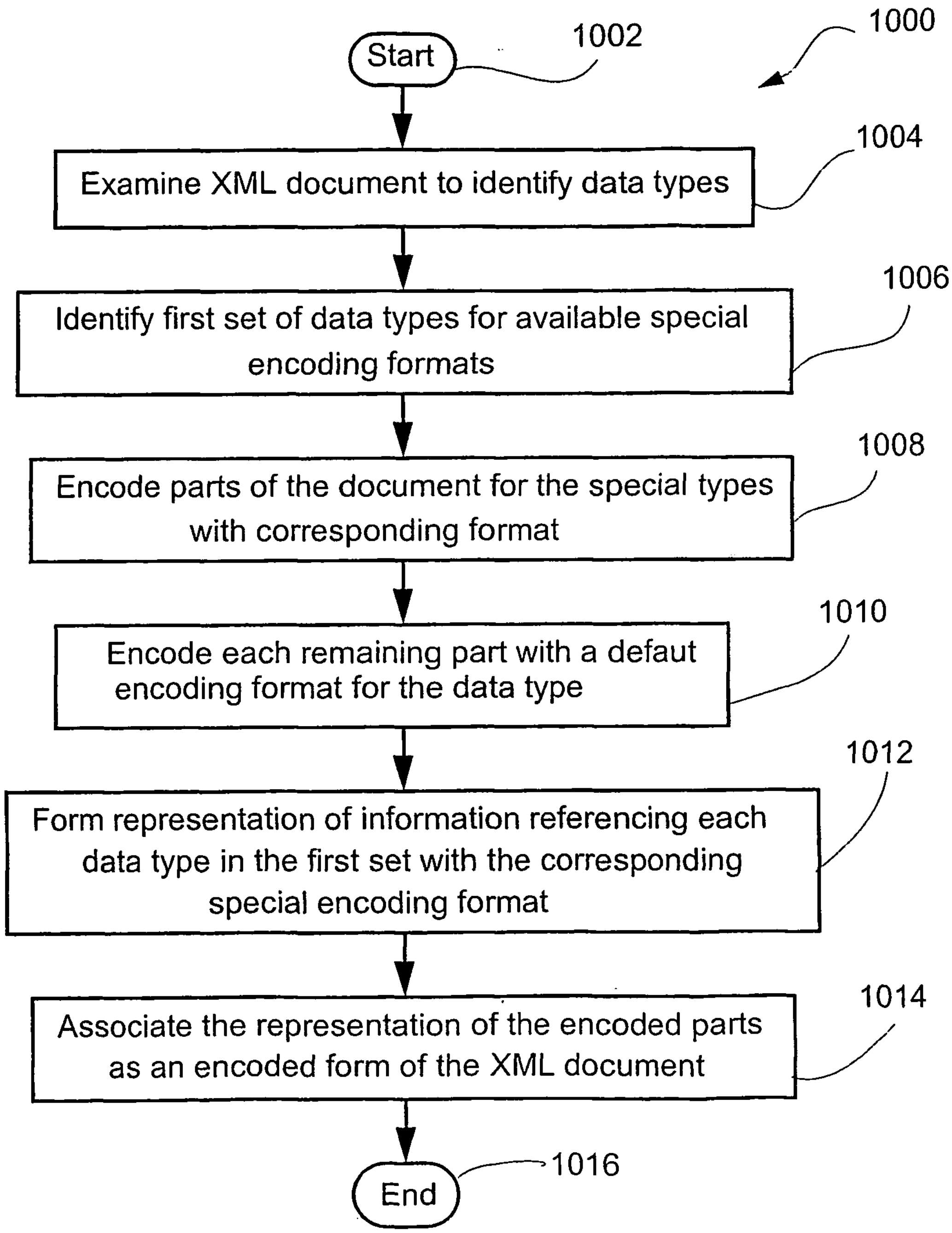

1000
Start
1002
Examine XML document to identify data types
1004
Identify first set of data types for available special encoding formats
1006
Encode parts of the document for the special types with corresponding format
1008
Encode each remaining part with a defaut encoding format for the data type
1010
Form representation of information referencing each data type in the first set with the corresponding special encoding format
1012
Associate the representation of the encoded parts as an encoded form of the XML document
1014
End
1016

Fig. 10

INTERNATIONAL SEARCH REPORT

International application No.
PCT/AU01/01257

A. CLASSIFICATION OF SUBJECT MATTER

Int. Cl. 7: G06F 17/00, H04N 7/26

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
WPAT,IEEE (stream+, document, hierarch+, link)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 00/33197 (Kent Ridge Digital Labs) 8 June 2000 Whole document	1,23,25,28,46
Y	"Compression of SMIL Documents" (Teng) IEEE Data Compression Conf. 28-30 March 2000 Whole document	48-52
Y	Research Disclosure (Derwent Abstract RD 430188) PAN 00-291162 (IBM Corp.) 10 February 2000 Abstract	48-52

[X] Further documents are listed in the continuation of Box C [X] See patent family annex

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
7 November 2001	13 NOV 2001
Name and mailing address of the ISA/AU AUSTRALIAN PATENT OFFICE PO BOX 200, WODEN ACT 2606, AUSTRALIA E-mail address: pct@ipaustralia.gov.au Facsimile No. (02) 6285 3929	Authorized officer **DALE E. SIVER** Telephone No : (02) 6283 2196

C (Continuation). **DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	EP 0 973 129 A2 (Matsushita Electric Industrial Co., Ltd.) 19 January 2000 Abstract, figures, claims	48-52
A	WO 98/23093 (NDS Limited) 28 May 1998 Abstract, claims	1-52
A	US 5 790 196 (Sun et al.) 4 August 1998 Abstract,figures	1-52
A	JP 2000 259667 (Derwent Abstract PAN 00-633611) 22 September 2000 Abstract	1

INTERNATIONAL SEARCH REPORT Information on patent family members	International application No. **PCT/AU01/01257**

This Annex lists the known "A" publication level patent family members relating to the patent documents cited in the above-mentioned international search report. The Australian Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

Patent Document Cited in Search Report		Patent Family Member			
WO	2000/33197	AU	15174/99		
EP	973129	CN	1250197	JP	2000/149039
WO	98/23093	EP	940043	GB	9624001
US	5790196	US	5969764	EP	892555

END OF ANNEX